QExPy Documentation

Release 2.0.2

Astral Cai, Connor Kapahi, Prof. Ryan Martin

Mar 01, 2023

CONTENTS:

CHAPTER

ONE

INTRODUCTION

QExPy (Queen's Experimental Physics) is a Python 3 package designed to facilitate data analysis in undergraduate physics laboratories. The package contains a module to easily propagate errors in uncertainty calculations, and a module that provides an intuitive interface to plot and fit data. The package is designed to be efficient, correct, and to allow for a pedagogic introduction to error analysis. The package is extensively tested in the Jupyter Notebook environment to allow high quality reports to be generated directly from a browser.

Highlights:

- Easily propagate uncertainties in calculations involving measured quantities
- Compare different methods of error propagation (e.g. Quadrature errors, Monte Carlo errors)
- Correctly include correlations between quantities when propagating uncertainties
- Calculate derivatives of calculated values with respect to the measured quantities from which the value is derived
- Flexible display formats for values and their uncertainties (e.g. number of significant figures, different ways of displaying units, scientific notation)
- Smart unit tracking in calculations (in development)
- Fit data to common functions (polynomials, gaussian distribution) or any custom functions specified by the user
- Intuitive interface for data plotting built on matplotlib

CHAPTER

TWO

GETTING STARTED

[1]:
```
import qexpy as q
```

The core of QExPy is a data structure called `ExperimentalValue`, which represents a value with an uncertainty. Any measurements recorded with QExPy and the result of any data analysis done with these measurements will all be wrapped in an instance of this class.

[2]:
```
# a measurement can be taken with a value, an uncertainty, a unit and a name
# (the last two are optional)
m = q.Measurement(15, 0.5, unit="kg", name="mass")
print(m)
```

```
mass = 15.0 +/- 0.5 [kg]
```

[3]:
```
# multiple measurements can be taken towards the same quantity
t = q.Measurement([5, 4.9, 5.3, 4.7, 4.8, 5.3], unit="s", name="time")

# this measurement will contain some basic statistic properties
print("Mean: {}".format(t.mean))
print("Error on the mean: {}".format(t.error_on_mean))
print("Standard deviation: {}".format(t.std))
```

```
Mean: 5.0
Error on the mean: 0.10327955589886441
Standard deviation: 0.2529822128134702
```

[4]:
```
# by default, the mean and error on the mean are used for this measurement
print(t)
```

```
time = 5.0 +/- 0.1 [s]
```

[5]:
```
# however, you can change that if you want
t.use_std_for_uncertainty()
print(t)
```

```
time = 5.0 +/- 0.3 [s]
```

[6]:
```
# let's do some calculations with measurements
vi = q.Measurement(0, unit="m/s", name="initial speed")
vf = q.Measurement(10, 0.5, unit="m/s", name="final speed")
a = (vf - vi) / t  # acceleration
a.name = "acceleration"
print(a)
```

```
acceleration = 2.0 +/- 0.1 [ms^-2]
```

[7]:
```
f = m * a
f.name = "force"
# as you can see below, the errors as well as units are propagated properly.
print(f)
```

```
force = 30 +/- 2 [kgms^-2]
```

2.1 Methods of Error Propagation

There are two error methods supported by QExPy.

2.1.1 Derivative Method (default)

By default, QExPy propagates the uncertainties using the "derivative" method. That is, for a function, $f(x, y)$, that depends on measured quantities $x \pm \sigma_x$ and $y \pm \sigma_y$, with covariance σ_{xy} between the two measured quantities, the uncertainty in f is given by:

$$\sigma_f = \sqrt{\left(\frac{\partial f}{\partial x}\sigma_x\right)^2 + \left(\frac{\partial f}{\partial y}\sigma_y\right)^2 + 2\frac{\partial f}{\partial x}\frac{\partial f}{\partial y}\sigma_{xy}}$$

Although the derivative method is commonly taught in undergraduate laboratories, it is only valid when the relative uncertainties in the quantities being propagated are small (e.g. less than ~10% relative uncertainty). This method is thus not strongly encouraged, although it has been made the default because it is so prevalent in undergraduate teaching.

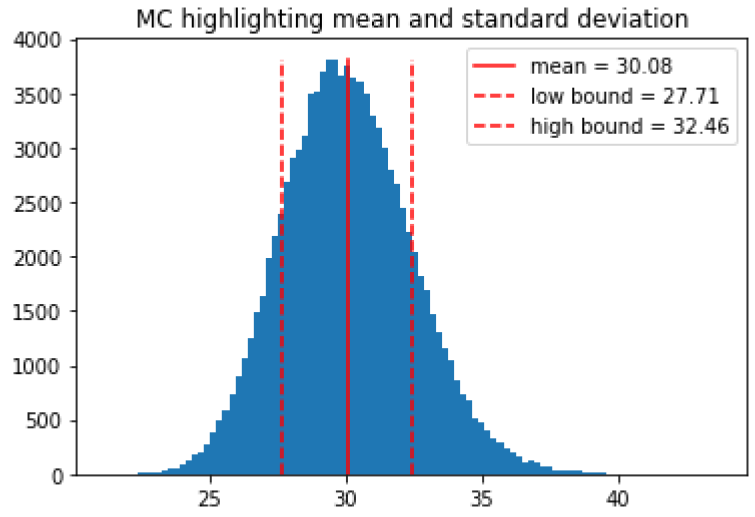
2.1.2 Monte Carlo Method (recommanded)

The MC method is based on a statiscal understanding of the measurements. In the QExPy implementation, currently, the main assumptions is that the uncertainty in a quantity is given by a "standard error"; that is, if $x = 10 \pm 1$, then we *assume* that this error and uncertainty should be interpreted as: "if we measure x multiple times, we will obtain a set of measurements that are normally distributed with a mean of 10 and a standard deviation of 1". In other words, we assume that x has a 68% chance of being in the range between 9 and 11.

The MC method then uses the assumption that measured quantities are normally distributed and use this to propagate the errors by using Monte Carlo simulation. Suppose that we have measured x and y and wish to determine the central value and uncertainty in $x = x + y$. The Monte Carlo method will generate normally distributed random values for x and y (the random numbers will be correctly correlated if the user has indicated that x and y are correlated), then it will add those random values together, to obtain a set of values for z. The mean and standard deviation of the random values for z are taken as the central value and uncertainty in z.

[8]:
```
# first change the number of significant figures so that we can see the difference
q.set_sig_figs_for_error(3)

# you can change the error method
q.set_error_method(q.ErrorMethod.MONTE_CARLO)
print(f)

# you can see the histogram of samples from the Monte Carlo simulation
f.mc.show_histogram()
```

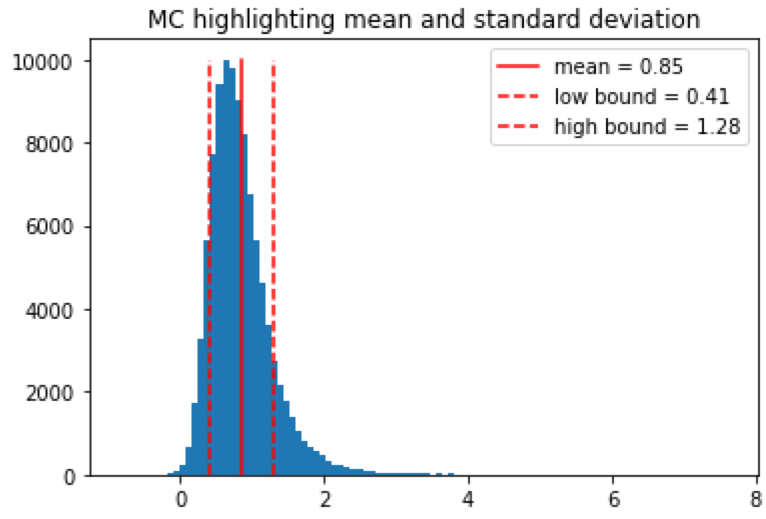
```
force = 30.08 +/- 2.38 [kgms^-2]
```



In the above example, the result of a Monte Carlo simulation has a perfect Gaussian distribution. However, this might not always be the case.

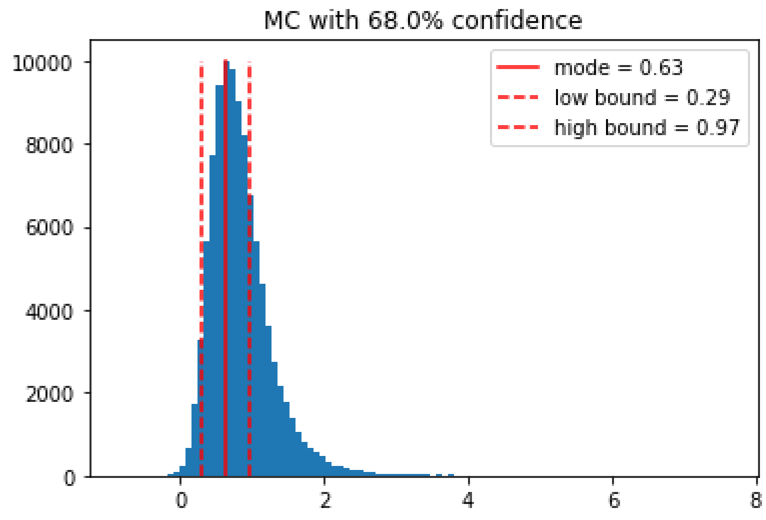
```
[9]: # let's try calculating the gravitational force between two stars

G = 6.67384e-11  # the gravitational constant
m1 = q.Measurement(40e4, 2e4, name="m1", unit="kg")
m2 = q.Measurement(30e4, 10e4, name="m2", unit="kg")
r = q.Measurement(3.2, 0.5, name="distance", unit="m")

f = G * m1 * m2 / (r ** 2)
print(f)

f.mc.show_histogram()
```

```
0.848 +/- 0.436 [kg^2m^-2]
```



As you can see, in this case, the mean and standard deviation of the distribution doesn't quite capture the center value and uncertainty we are looking for. We can try a different strategy where we find the mode (most probably value) of the distribution, and a confidence range.

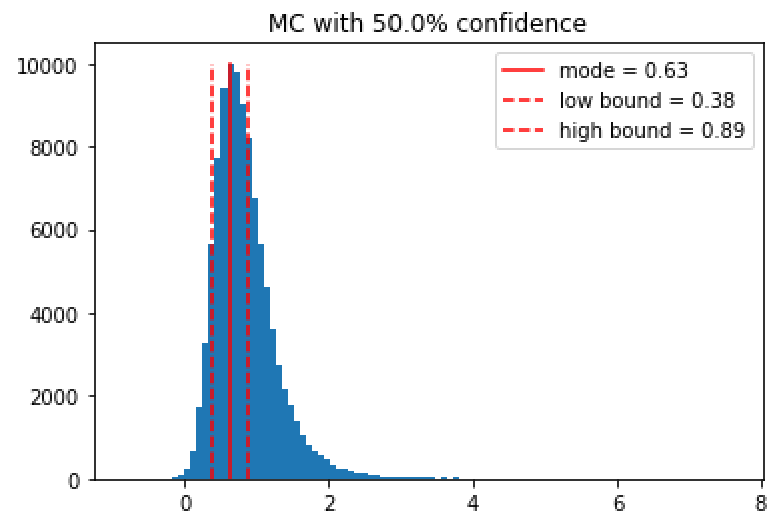
```
[10]: # change the mc strategy
f.mc.use_mode_with_confidence()

# show the histogram again
f.mc.show_histogram()

# also print the value
print(f)
```



```
0.632 +/- 0.339 [kg^2m^-2]
```

By default, the confidence level is 68%, but we can change that.

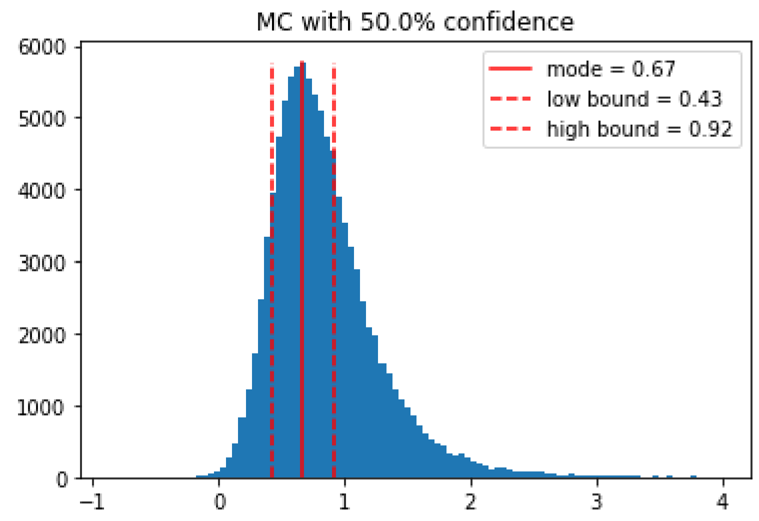
```
[11]: # change the confidence
f.mc.confidence = 0.5

# show the histogram again
f.mc.show_histogram()

# also print the value
print(f)
```



```
0.632 +/- 0.254 [kg^2m^-2]
```

As you can see, most of the samples are concentrated in the first half of the histogram. In order to increase resolution, the user can manually set the range of the histogram to focus on the region with the most samples.

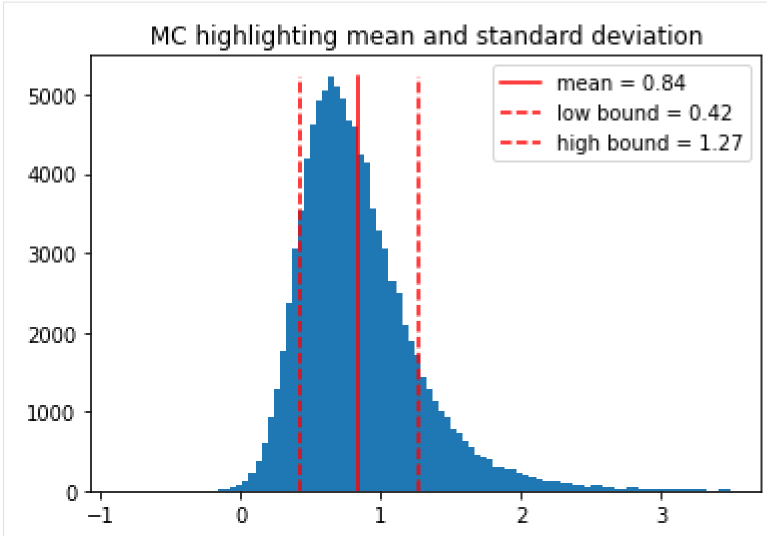
```
[12]: # try show histogram again
f.mc.show_histogram(range=(-1,4))
```



If you feel like this subset of the distribution is somewhat more representative of the quantity, you can set the range for Monte Carlo simulation to this interval

```
[13]: # try setting the range
f.mc.set_xrange(-1, 3.5)
f.mc.use_mean_and_std()  # let's see what the mean and std is now
print(f)
f.mc.show_histogram()
```

```
0.844 +/- 0.421 [kg^2m^-2]
```



2.2 Print Style and Formatting

[14]:
```
# you can specify the precision of the values
q.set_sig_figs_for_value(3)
print("{} (the value has 3 significant figures)".format(f))
q.set_sig_figs_for_error(3)
print("{} (the uncertainty has 3 significant figures)".format(f))
```

```
0.844 +/- 0.421 [kg^2m^-2] (the value has 3 significant figures)
0.844 +/- 0.421 [kg^2m^-2] (the uncertainty has 3 significant figures)
```

[15]:
```
# you can change the print formatting
q.set_print_style(q.PrintStyle.SCIENTIFIC)
print(f)
q.set_print_style(q.PrintStyle.LATEX)
print(f)
# or reset it to default
q.set_print_style(q.PrintStyle.DEFAULT)
print(f)
```

```
(8.44 +/- 4.21) * 10^-1 [kg^2m^-2]
(8.44 \pm 4.21) * 10^-1 [kg^2m^-2]
0.844 +/- 0.421 [kg^2m^-2]
```

[16]:
```
# you can change the style of how units are displayed
q.set_unit_style(q.UnitStyle.EXPONENTS)  # this is the default
print("default (exponent) style unit printing: {}".format(f.unit))
q.set_unit_style(q.UnitStyle.FRACTION)  # more intuitive but sometimes ambiguous
print("fraction style unit printing: {}".format(f.unit))

# print the complete value
print("\nThe complete value representation will change accordingly:\n{}".format(f))
```

```
default (exponent) style unit printing: kg^2m^-2
fraction style unit printing: kg^2/m^2

The complete value representation will change accordingly:
0.844 +/- 0.421 [kg^2/m^2]
```

2.3 Correlated Measurements

Sometimes when two series of measurements are correlated, the error propagation should reflect that. It's worth noting that repeated measurements of the same length are not automatically correlated. Whether two measurements are physically correlated is at the discretion of the user. There are two values related to correlated measurements. The "covariance" indicates the extent to which two random variables change in tandem, and "correlation" is indicates how strongly two variables are related, which is confined between -1 and 1

[17]:
```
q.reset_default_configuration()  # first reset everything to default
q.set_sig_figs_for_error(3)

# first let's take two series of measurements
m1 = q.Measurement([20, 20.2, 20.3, 20.4])
m2 = q.Measurement([20, 20.1, 19.8, 20.3])

result = m1 + m2
print(result)
```

```
40.275 +/- 0.135
```

[18]:
```
# let's say m1 and m2 are measured together, and they might be correlated
q.set_correlation(m1, m2)  # this declares the correlation

# since m1 and m2 are of the same length, QExPy is able to calculate the covariance
cor = q.get_correlation(m1, m2)
cov = q.get_covariance(m1, m2)
print("Covariance: {}\nCorrelation: {}".format(cov, cor))
```

```
Covariance: 0.011666666666666523
Correlation: 0.3281650616569432
```

[19]:
```
# as a result, the error will be recalculated
result.recalculate()  # since something changed, this ensures that everything is updated
print(result)
```

```
40.275 +/- 0.155
```

[20]:
```
# the monte carlo simulated results will also be properly correlated
result.recalculate()
q.set_error_method(q.ErrorMethod.MONTE_CARLO)
print(result)
```

```
40.272 +/- 0.153
```

[21]:
```
# the user can personally set the correlation between two values
q.set_correlation(m1, m2, 0.8)
```

(continues on next page)

(continued from previous page)

```
# q.set_covariance(m1, m2, 0.02)

result.recalculate()  # first ask that the value is updated

q.set_error_method(q.ErrorMethod.DERIVATIVE)
print(result)
q.set_error_method(q.ErrorMethod.MONTE_CARLO)
print(result)
```

```
40.275 +/- 0.180
40.276 +/- 0.180
```

2.4 Measurement Arrays

QExPy can also handle a series of measurements with `MeasurementArray`, which is an array of individual measurements, each with an uncertainty. This is a sub-class of `numpy.ndarray`, so it can be operated on as one.

[22]:
```
q.reset_default_configuration()
q.set_sig_figs_for_error(2)

# you can record an array with the same uncertainty throughout
arr1 = q.MeasurementArray([1, 2, 3, 4, 5], 0.5, name="length", unit="m")
print(arr1)
```

```
length = [ 1.00 +/- 0.50, 2.00 +/- 0.50, 3.00 +/- 0.50, 4.00 +/- 0.50, 5.00 +/- 0.50 ]␣
↪(m)
```

[23]:
```
# if the error is left out, it's by default set to 0
arr2 = q.MeasurementArray([1, 2, 3, 4, 5])
print(arr2)
```

```
[ 1.0 +/- 0, 2.0 +/- 0, 3.0 +/- 0, 4.0 +/- 0, 5.0 +/- 0 ]
```

[24]:
```
# you can record an array of measurement with distinct uncertainties
arr3 = q.MeasurementArray(
    [1, 2, 3, 4, 5], [0.1, 0.2, 0.3, 0.4, 0.5], name="length", unit="m")
print(arr3)
```

```
length = [ 1.00 +/- 0.10, 2.00 +/- 0.20, 3.00 +/- 0.30, 4.00 +/- 0.40, 5.00 +/- 0.50 ]␣
↪(m)
```

[25]:
```
# individual measurements can be extracted
measurement = arr3[2]
print(measurement)
```

```
length_2 = 3.00 +/- 0.30 [m]
```

[26]:
```
# the measurement array has basic statistical uncertainties
print("Mean: {}".format(arr3.mean()))
print("Sum: {}".format(arr3.sum()))
print("Standard Deviation: {}".format(arr3.std()))
print("Error Weighted Mean: {}".format(arr3.error_weighted_mean()))
```

```
Mean: mean of length = 3.00 +/- 0.71 [m]
Sum: length = 15.00 +/- 0.74 [m]
Standard Deviation: 1.5811388300841898
Error Weighted Mean: 1.5600683241601823
```

[27]:
```
# basic operations with measurement arrays
arr4 = arr3.delete(0)
arr4.name = "arr4"
print(arr4)
arr5 = arr4.append((6, 0.6))
arr5.name = "arr5"
print(arr5)
# operations can be chained
arr6 = arr4.delete(0).append([(6, 0.6),(7, 0.7)])
arr6.name = "arr6"
print(arr6)
arr7 = arr4.insert(0, (6, 0.6))
arr7.name = "arr7"
print(arr7)
```

```
arr4 = [ 2.00 +/- 0.20, 3.00 +/- 0.30, 4.00 +/- 0.40, 5.00 +/- 0.50 ] (m)
arr5 = [ 2.00 +/- 0.20, 3.00 +/- 0.30, 4.00 +/- 0.40, 5.00 +/- 0.50, 6.00 +/- 0.60 ] (m)
arr6 = [ 3.00 +/- 0.30, 4.00 +/- 0.40, 5.00 +/- 0.50, 6.00 +/- 0.60, 7.00 +/- 0.70 ] (m)
arr7 = [ 6.00 +/- 0.60, 2.00 +/- 0.20, 3.00 +/- 0.30, 4.00 +/- 0.40, 5.00 +/- 0.50 ] (m)
```

[28]:
```
# the index of variables are calculated accordingly
for element in arr7:
    print(element)
```

```
arr7_0 = 6.00 +/- 0.60 [m]
arr7_1 = 2.00 +/- 0.20 [m]
arr7_2 = 3.00 +/- 0.30 [m]
arr7_3 = 4.00 +/- 0.40 [m]
arr7_4 = 5.00 +/- 0.50 [m]
```

2.5 Math Functions

QExPy includes wrappers for some basic math functions, which works on both individual values and measurement arrays. Available functions include basic trig functions such as `sin`, `cos`, `tan`, and `sec`, `csc`, and other functions such as `sqrt`, `log`, `exp`.

[29]:
```
# for example, try finding out the time it takes for an object to fall 8 meters
h = q.Measurement(8,0.1)
g = 9.81
t = q.sqrt(2*h/g)
t.name = "time"
t.unit = "s"
print(t)
```

```
time = 1.2771 +/- 0.0080 [s]
```

```
[30]: # the log function can take 1 or 2 arguments
res = q.log(h)  # by default the base is e
print("log base e of h: {}".format(res))
res = q.log(2, h)  # you can specify the base
print("log base 2 of h: {}".format(res))
```

```
log base e of h: 2.079 +/- 0.012
log base 2 of h: 3.000 +/- 0.018
```

Right now unit propagation with functions is not yet supported. It will be implemented in future versions

CHAPTER

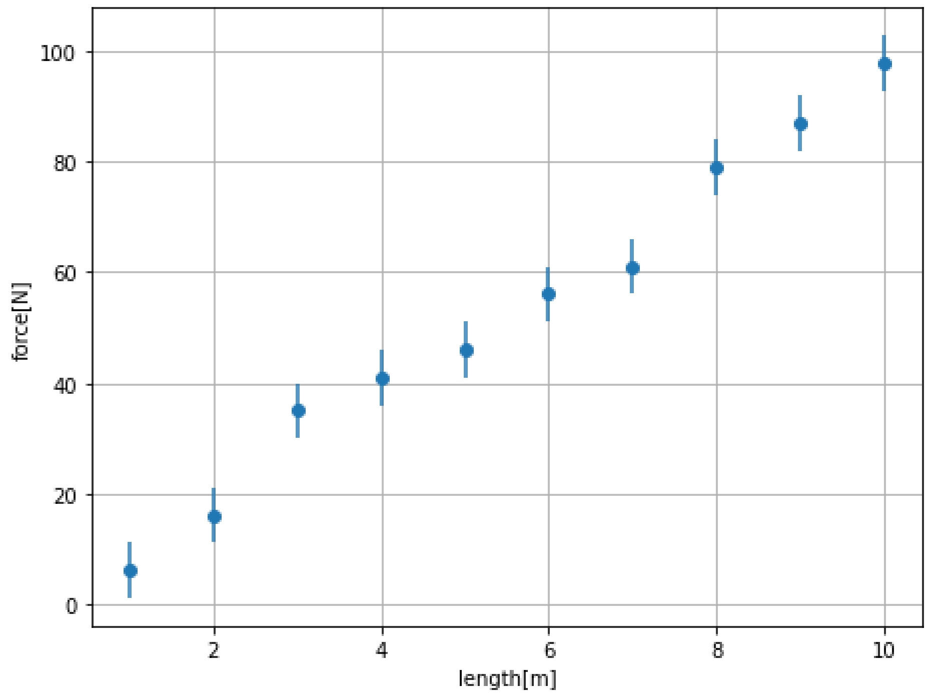
THREE

INTRO TO PLOTTING AND FITTING

```
[1]: import qexpy as q
     import qexpy.plotting as plt
```

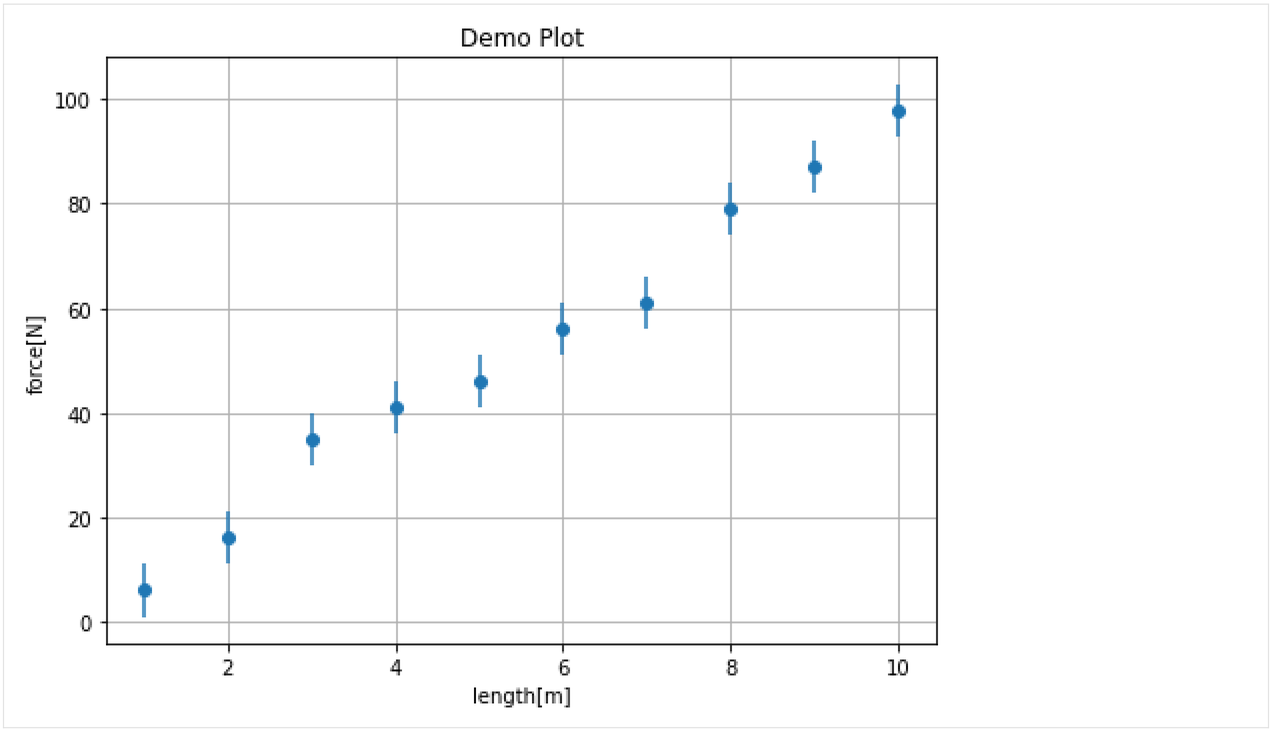
The plotting module of QExPy is a wrapper for matplotlib.pyplot, developed to interface with QExPy data structures.

```
[2]: # let's start by creating some arrays of measurement
     xdata = q.MeasurementArray(
         [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], unit="m", name="length")
     ydata = q.MeasurementArray(
         [6, 16, 35, 41, 46, 56, 61, 79, 87, 98], error=5, unit="N", name="force")

     # now we can add them to a simple plot
     plt.plot(xdata, ydata, name="first")
     # use `figure = plt.plot(xdata, ydata)` to obtain the Plot object instance for further
     # customization. qexpy.plotting keeps a buffer of the latest Plot instance, if you did
     # not assign the return value of plt.plot to anything (like what we are doing here), you
     # can still retrieve the Plot instance using `figure = plt.get_plot()`, as shown below.

     # draw the plot to the screen
     plt.show()
```

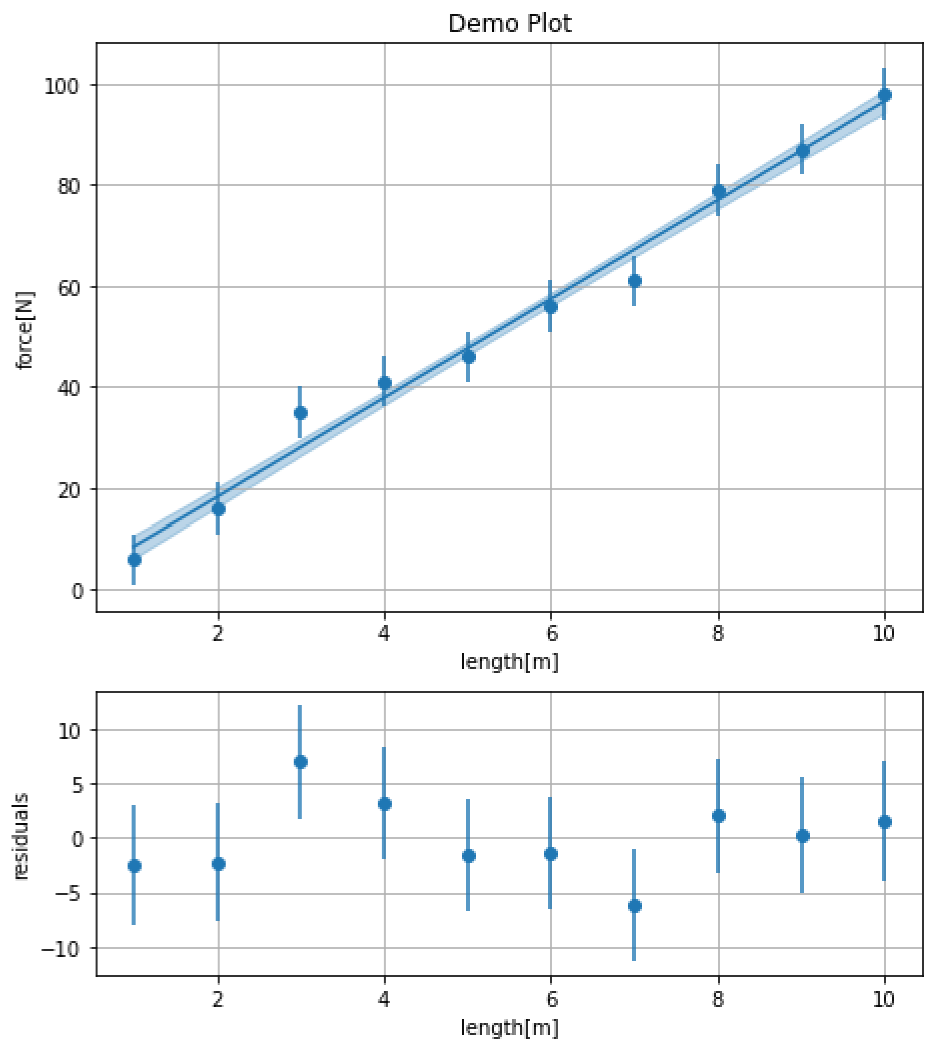


As you can see in the plot above, the name and units of the data that's passed in are automatically added to the plot as axis labels. For simple plotting purposes, this is enough. However, if you wish to further customize the plot, you can try to operate directly on the plot object. You will be able to change the title as well as the axis labels yourself. You can also add error bars and legends to the plot.

```
[3]: # retrieve the current plot object
figure = plt.get_plot()

# As you can see, the error bars are automatically on.
# If not, we can manually add error bars to the plot
figure.error_bars()  # use `figure.error_bars(false)` to turn off error bars

# we can add a title to the plot
figure.title = "Demo Plot"

# finally draw the plot
figure.show()
```



```
[4]: # We can try to add a fit to the plot. The fit function automatically selects the last
     # applicable fit target (a data set or a histogram) on the plot.
     result = figure.fit(model=q.FitModel.LINEAR)

     # also add a residuals subplot
     figure.residuals()

     # show the plot and the result
     figure.show()
     print(result)
```



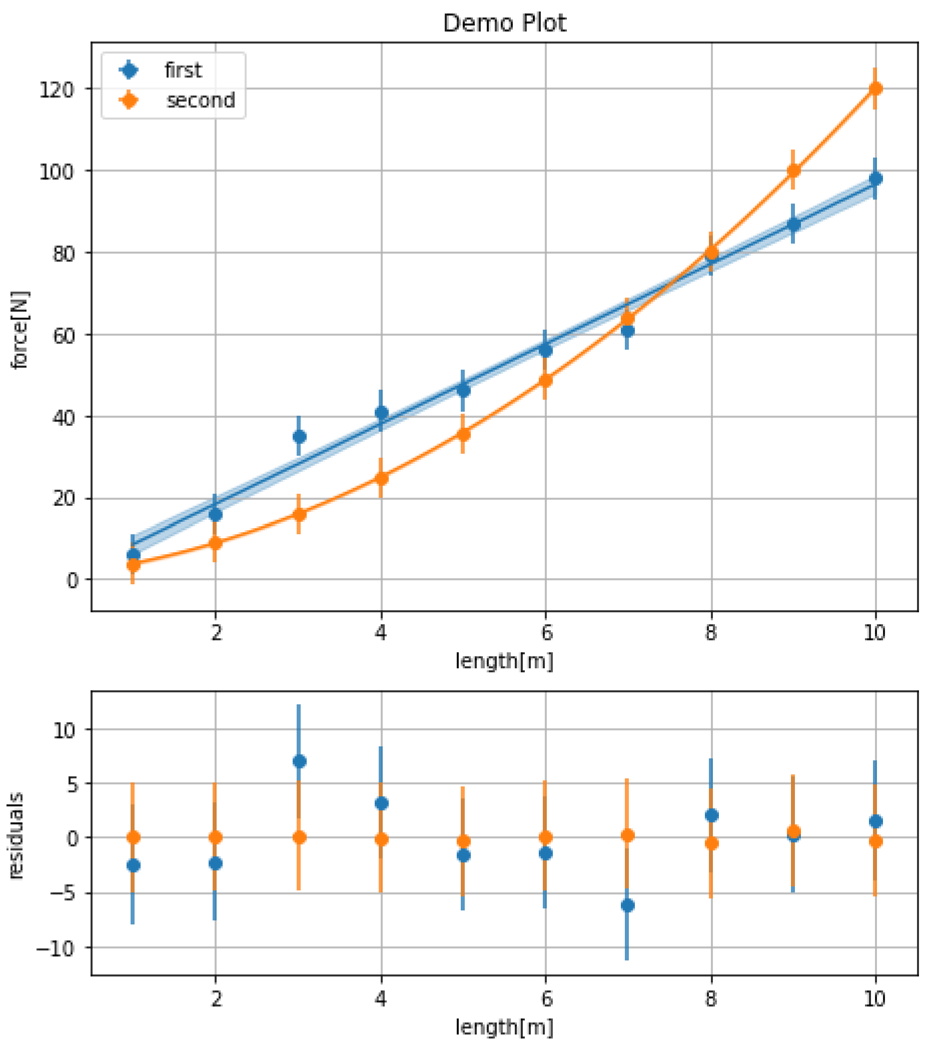
```
----------------- Fit Results -------------------
Fit of first to linear

Result Parameter List:
slope = 9.8 +/- 0.4,
intercept = -1 +/- 3

Correlation Matrix:
[[ 1.    -0.886]
 [-0.886  1.   ]]

chi2/ndof = 4.75/7

--------------- End Fit Results -----------------
```

[5]:
```
# we can add multiple datasets to plot
figure.plot(xdata=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
            ydata=[3.8, 8.9, 16, 24.8, 35.5, 48.9, 64, 80, 100, 120],
            xerr=0.05, yerr=5, name="second")
```

(continues on next page)

(continued from previous page)

```
# we can also add a line of best fit to the plot
figure.fit(model=q.FitModel.QUADRATIC)

# we can add turn on legends for the plot since we now have 2 data sets
figure.legend()

# now show the figure
figure.show()
```



3.1 The Fitting Module

The QExPy fitting module supports a few pre-set fit models, as well as any custom fit function the user wish to use. The available pre-set models include linear fit, quadratic fit, general polynomial fit, exponential fit, and gaussian fit. The pre-set models are stored under q.FitModel, To select the fit model, if you're in a Jupyter Notebook environment, simply type "q.FitModel.", and press TAB, the available options will appear as a list of autofill suggestions

[6]:
```
# We can do a simple quadratic fit.
result = q.fit(
    xdata=[1,2,3,4,5,6,7,8,9,10], xerr=0.5,
    ydata=[3.86,8.80,16.11,24.6,35.71,48.75,64,81.15,99.72,120.94],
    yerr=0.5, model=q.FitModel.QUADRATIC)  # or simply type "quadratic"

print(result)
```

```
----------------- Fit Results -------------------
Fit of XY Dataset to quadratic

Result Parameter List:
a = 1.004 +/- 0.009,
b = 2.0 +/- 0.1,
c = 0.9 +/- 0.2

Correlation Matrix:
[[ 1.    -0.975  0.814]
 [-0.975  1.    -0.909]
 [ 0.814 -0.909  1.   ]]

chi2/ndof = 1.13/6

--------------- End Fit Results -----------------
```

The parameters of polynomials are organized from highest to lowest power terms. The result above indicates that the function of best fit is 1.004x^2 + 2x + 0.9

The QExPy fit function is very flexible in accepting fit arguments. The three accepted ways to specify the fit data set are: 1. Create an XYDataSet object and pass the dataset into the fit function 2. Pass in a MeasurementArray object for each of xdata and ydata 3. Pass in two Python lists or numpy arrays for xdata and ydata, specify the xerr or yerr if applicable

[7]:
```
# The traditionoal way (with previous versions of QExPy) of fitting
xydata = q.XYDataSet(
    xdata=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], xname='length', xunit='m',
    ydata=[0.6, 1.6, 3.5, 4.1, 4.6, 5.6, 6.1, 7.9, 8.7, 9.8], yerr=0.5,
    yname='force', yunit='N')

# the fit function can be called directly from the data set
result = xydata.fit("linear")

print(result)
```

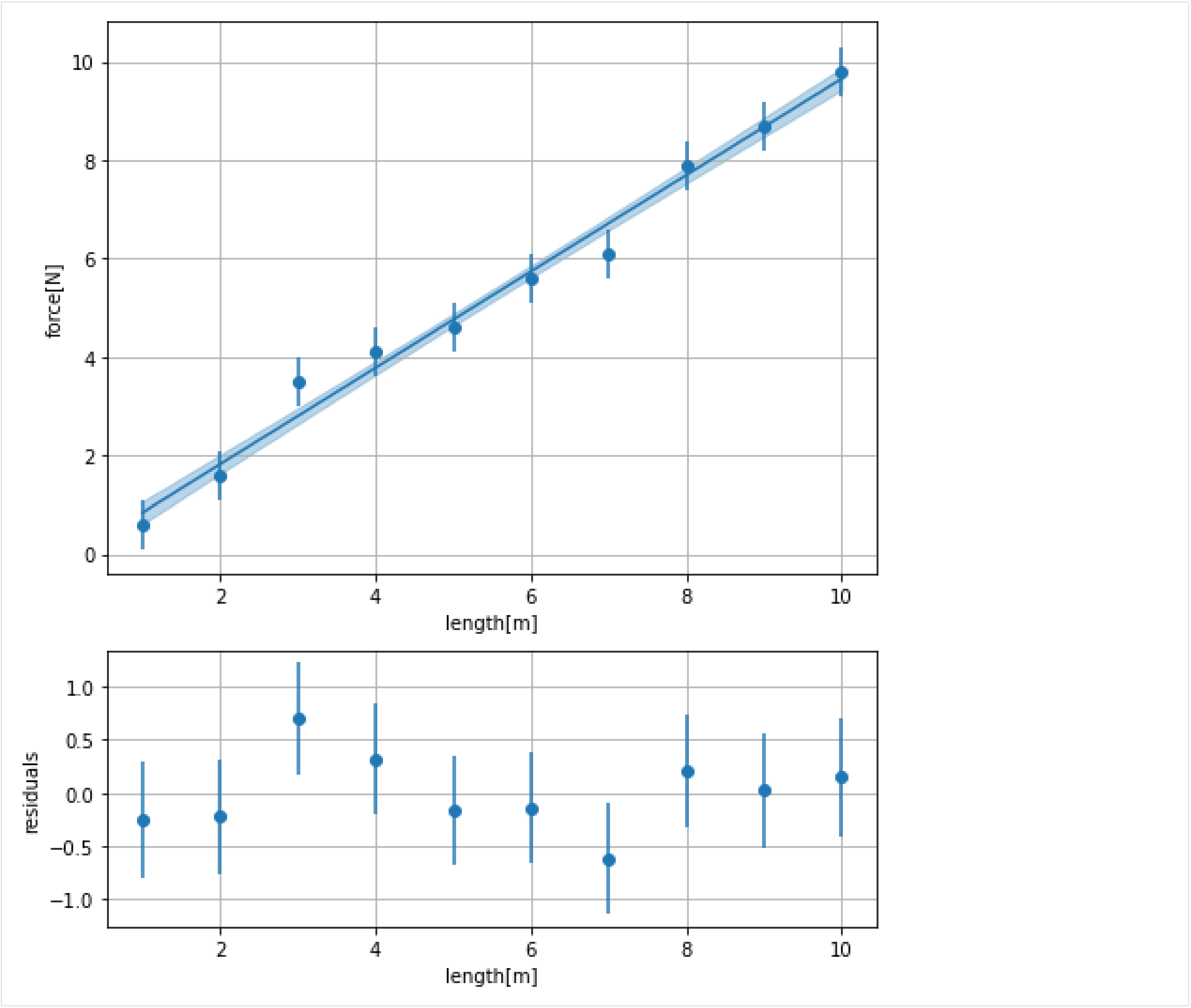
```
----------------- Fit Results -------------------
Fit of XY Dataset to linear
```

(continues on next page)

(continued from previous page)

```
Result Parameter List:
slope = 0.98 +/- 0.04,
intercept = -0.1 +/- 0.3

Correlation Matrix:
[[ 1.    -0.886]
 [-0.886  1.   ]]

chi2/ndof = 4.75/7

--------------- End Fit Results -----------------
```

[8]:
```
# You can very easily add a dataset and its fit function to a plot
figure = plt.plot(xydata)
figure.plot(result)

# turn on residuals and error bars
figure.error_bars()
figure.residuals()

# show the figure
figure.show()
```



```
[9]: # You can access the fit parameters easily by indexing the result instance:
slope = result[0]
intercept = result[1]

# these are both ExperimentalValue instances
print(slope)
print(intercept)
```

```
slope = 0.98 +/- 0.04
intercept = -0.1 +/- 0.3
```

```
[10]: # QExPy also supports fitting with higher order polynomials. The degree of a polynomial
# is the degree of the highest order term. e.g. a quadratic function would be degree-2
result = q.fit(
    xdata=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    ydata=[3.89, 18.01, 58.02, 135.92, 264.01, 453.99, 718.02, 1067.98, 1516.01, 2074],
    model=q.FitModel.POLYNOMIAL, degree=3)

print(result)
```

```
----------------- Fit Results -------------------
Fit of XY Dataset to polynomial

Result Parameter List:
coeffs_0 = 2.0006 +/- 0.0008,
coeffs_1 = 0.99 +/- 0.01,
coeffs_2 = -2.93 +/- 0.06,
coeffs_3 = 3.86 +/- 0.09

Correlation Matrix:
[[ 1.    -0.989  0.941 -0.795]
 [-0.989  1.    -0.979  0.859]
 [ 0.941 -0.979  1.    -0.935]
 [-0.795  0.859 -0.935  1.   ]]

chi2/ndof = 0.00/5

--------------- End Fit Results -----------------
```

3.1.1 Advanced Fitting

QExPy supports fitting a custom function to a data set. With any non-polynomial fit models, a list of guesses for the fit parameters needs to be supplied under the keyword argument "parguess". Other optional keyword arguments to the fit function includes "parnames" and "parunits", which are the names and units assigned to the fit parameters, which will show up in the fit results.

[11]:
```
# First define a fit model
def func(x, a, b):
    return a * q.sin(b * x)

# Apply it to the test dataset
result = q.fit(
    xdata=[0.00,0.33,0.66,0.99,1.32,1.65,1.98,2.31,2.64,2.97,3.31,3.64,3.97,4.30,
           4.63,4.96,5.29,5.62,5.95,6.28],
    ydata=[0.09,0.41,1.53,2.23,3.76,2.50,3.89,5.33,5.39,4.05,5.08,5.84,4.59,4.50,
           3.48,3.57,2.20,1.95,0.39,-0.18],
    model=func, parguess=[1, 1], parnames=["mass", "length"], parunits=["kg", "m"])

print(result)
```

```
----------------- Fit Results -------------------
Fit of XY Dataset to custom

Result Parameter List:
mass = 5.1 +/- 0.2 [kg],
length = 0.500 +/- 0.009 [m]

Correlation Matrix:
[[1.    0.238]
 [0.238 1.   ]]

chi2/ndof = 0.00/17
```

(continues on next page)

(continued from previous page)

```
---------------- End Fit Results ------------------
```

[12]:
```python
# The QExPy fitting module also has this little feature implemented, where if you leave
# the "parname" field empty, parameter names will be extracted from the signature.

def func(x, mass, length):  # define the fit function with the names you want
    return mass * q.sin(length * x)

# try the same fit again
result = q.fit(
    xdata=[0.00,0.33,0.66,0.99,1.32,1.65,1.98,2.31,2.64,2.97,3.31,3.64,3.97,4.30,
           4.63,4.96,5.29,5.62,5.95,6.28],
    ydata=[0.09,0.41,1.53,2.23,3.76,2.50,3.89,5.33,5.39,4.05,5.08,5.84,4.59,4.50,
           3.48,3.57,2.20,1.95,0.39,-0.18],
    model=func, parguess=[1, 1], parunits=["kg", "m"])

print(result)
```

```
----------------- Fit Results -------------------
Fit of XY Dataset to custom

Result Parameter List:
mass = 5.1 +/- 0.2 [kg],
length = 0.500 +/- 0.009 [m]

Correlation Matrix:
[[1.    0.238]
 [0.238 1.   ]]

chi2/ndof = 0.00/17

---------------- End Fit Results ------------------
```

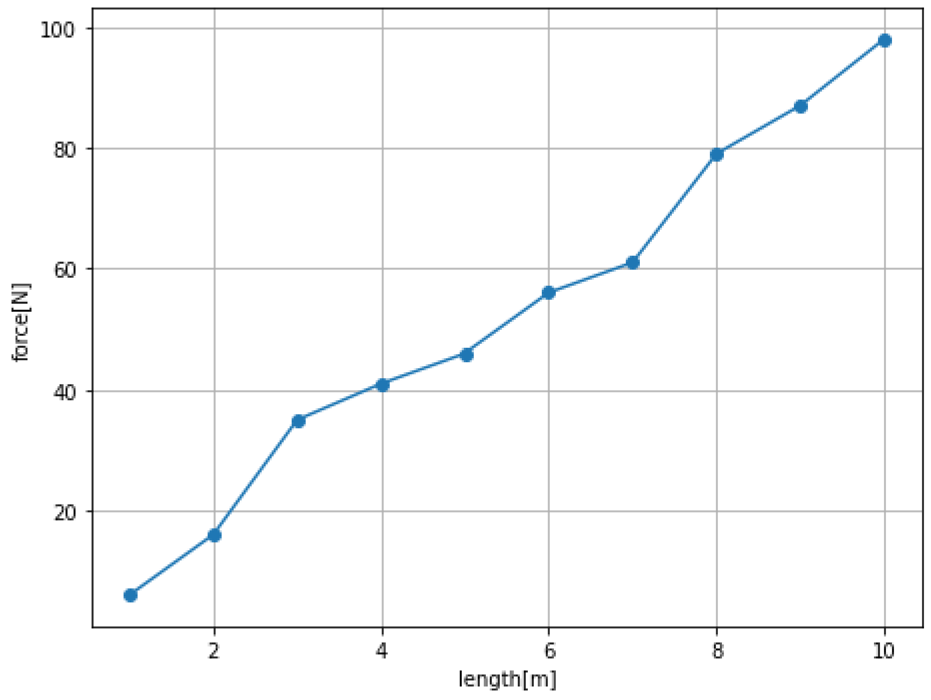
3.2 The Plotting Module

The plotting module is centered around the Plot class. It is a data structure that holds all the objects to be plotted (data sets, functions, histograms, etc.). When calling the QExPy plot function, a Plot object will be created and returned. The user can add objects to the plot using the same plot function called from the Plot instance. The module also keeps a reference to the last Plot object being operated on, and if the return value of a call to the plot function is not assigned to any variable, by default, the object will be added to the current buffered plot.

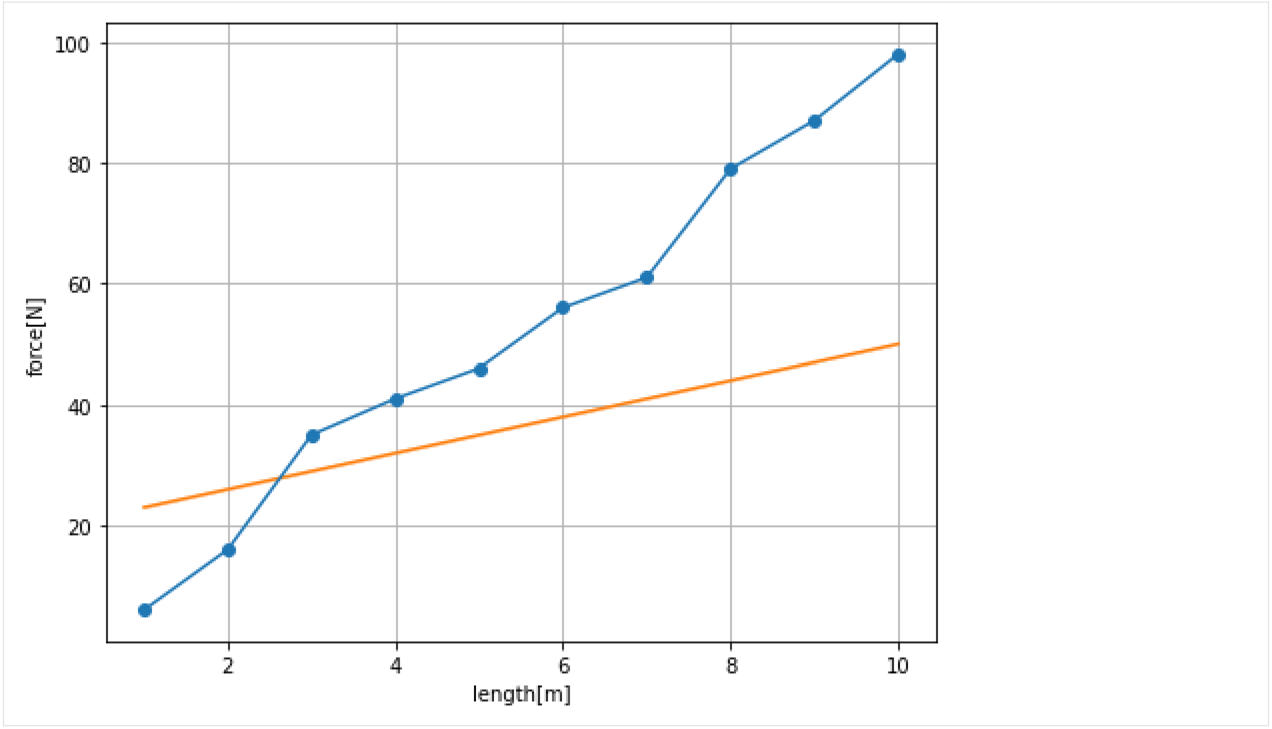
3.2.1 Plotting Data Sets and Functions

The QExPy plotting function takes the same types of inputs as the fit function for plotting data sets.

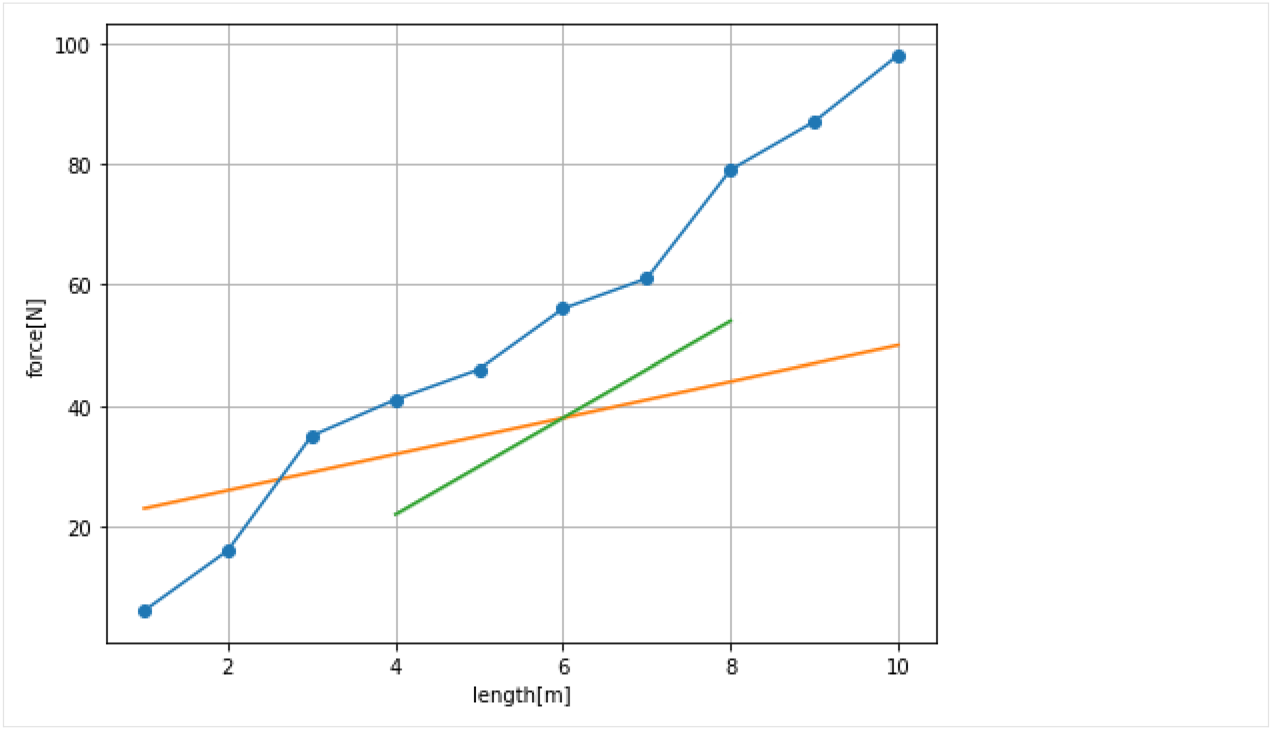
[13]:
```
xydata = q.XYDataSet(
    xdata=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], xname='length', xunit='m',
    ydata=[6, 16, 35, 41, 46, 56, 61, 79, 87, 98], yerr=0.5,
    yname='force', yunit='N')

# You can specify the format string of the plot object. The default for data sets is
# dots, but if you want them connected in a line, you can use the fmt option
figure = plt.plot(xydata, "-o")

figure.show()
```



[14]:
```
# You can also add callable functions to a Plot

def func(x):
    return 20 + x * 3

figure.plot(func)

figure.show()
```



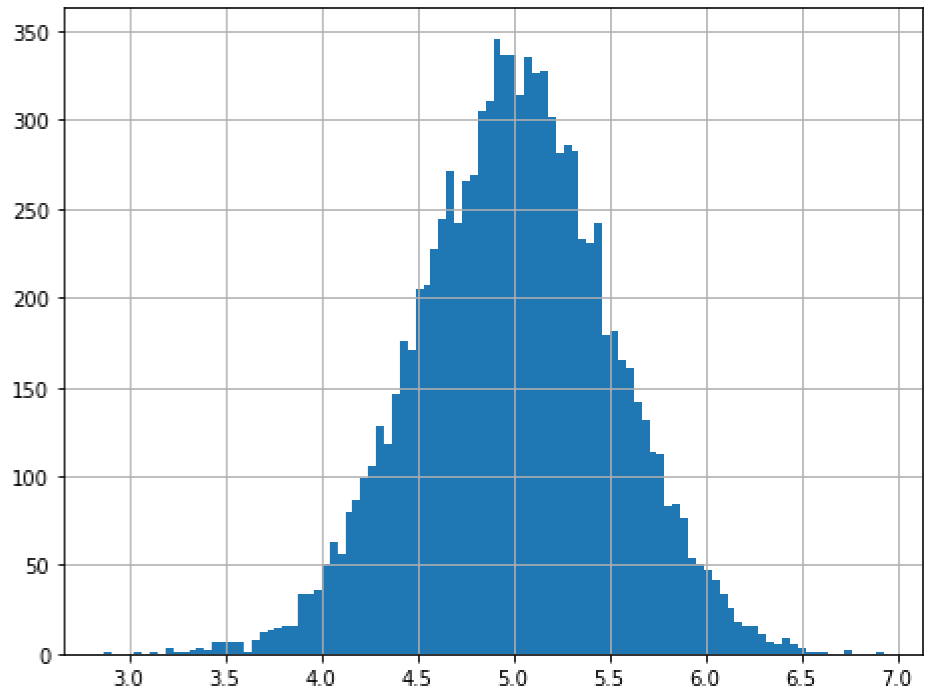
[15]:
```
# You can also plot a function with parameters
def func2(x, *pars):
    return pars[0] + x * pars[1]

# You can specify an xrange for the plot, and also when plotting a function with
# parameters, you have to specify the parameters values too.
figure.plot(func2, xrange=(4,8), pars=[-10,8])

figure.show()
```



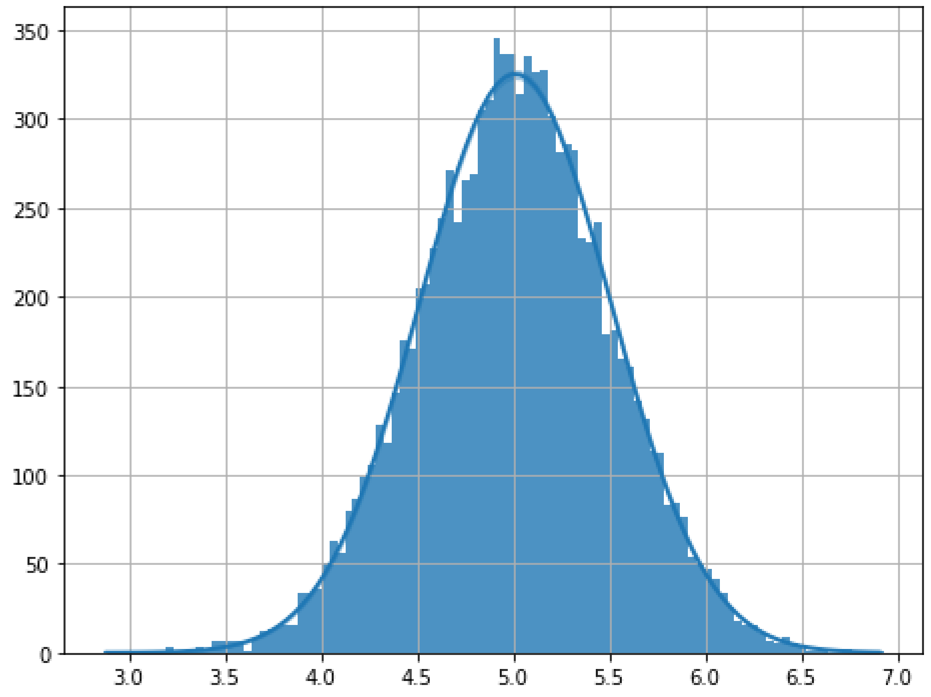
3.2.2 Plotting Histograms

The QExPy plotting module is also capable of plotting histograms.

```
[16]: # first let's generate a bunch of random numbers
import numpy as np
samples = np.random.normal(5, 0.5, 10000)

# Let's plot it out as a histogram. Note that the return values include the array
# of counts, the bin edges, followed by the Plot object.
n, bins, figure = plt.hist(samples, bins=100)

figure.show()
```



[17]:
```
# now let's try adding a fit to the histogram
result = figure.fit(model=q.FitModel.GAUSSIAN, parguess=[100, 5, 0.5])

figure.show()
print(result)
```



```
----------------- Fit Results -------------------
Fit of histogram to gaussian

Result Parameter List:
normalization = 405 +/- 3,
mean = 5.011 +/- 0.004,
std = 0.497 +/- 0.004

Correlation Matrix:
[[1.000e+00 1.785e-06 5.774e-01]
 [1.785e-06 1.000e+00 2.826e-06]
 [5.774e-01 2.826e-06 1.000e+00]]

chi2/ndof = 0.00/96

--------------- End Fit Results -----------------
```

CHAPTER

FOUR

THE EXPERIMENTALVALUE OBJECT

class qexpy.data.data.**ExperimentalValue**(*unit=''*, *name=''*, *save=True*)

Base class for quantities with a value and an uncertainty

The ExperimentalValue is a container for an individual quantity involved in an experiment and subsequent data analysis. Each quantity has a value and an uncertainty (error), and optionally, a name and a unit. Experimental-Value instances can be used in calculations just like any other numerical variable in Python. The result of such calculations will be wrapped in ExperimentalValue instances, with the properly propagated uncertainties.

Examples

```
>>> import qexpy as q
```

```
>>> a = q.Measurement(302, 5) # The standard way to initialize an ExperimentalValue
```

```
>>> # Access the basic properties
>>> a.value
302
>>> a.error
5
>>> a.relative_error  # This is defined as error/value
0.016556291390728478
```

```
>>> # These properties can be changed
>>> a.value = 303
>>> a.value
303
>>> a.relative_error = 0.05
>>> a.error  # The error and relative_error are connected
15.15
```

```
>>> # You can specify the name or the units of a value
>>> a.name = "force"
>>> a.unit = "kg*m^2/s^2"
```

```
>>> # The string representation of the value will include the name and units
>>> print(a)
force = 300 +/- 20 [kgm^2s^-2]
```

```
>>> # You can also specify how you want the values or the units to be printed
>>> q.set_print_style(q.PrintStyle.SCIENTIFIC)
>>> q.set_unit_style(q.UnitStyle.FRACTION)
>>> q.set_sig_figs_for_error(2)
>>> print(a)
force = (3.03 +/- 0.15) * 10^2 [kgm^2/s^2]
```

4.1 Properties

`ExperimentalValue.`**`value`**

The center value of this quantity

Type
float

`ExperimentalValue.`**`error`**

The uncertainty of this quantity

Type
float

`ExperimentalValue.`**`relative_error`**

The ratio of the uncertainty to its center value

Type
float

`ExperimentalValue.`**`name`**

The name of this quantity

Type
str

`ExperimentalValue.`**`unit`**

The unit of this quantity

Type
str

4.2 Methods

`abstract` `ExperimentalValue.`**`derivative`**(*other*)

Calculates the derivative of this quantity with respect to another

The derivative of any value with respect to itself is 1, and for unrelated values, the derivative is always 0. This method is typically called from a DerivedValue, to find out its derivative with respect to one of the measurements it's derived from.

Parameters
`other` (`ExperimentalValue`) – the target for finding the derivative

Return type
`float`

`ExperimentalValue.`**`get_covariance`**`(`*other*`)`

Gets the covariance between this value and another value

Return type
`float`

`ExperimentalValue.`**`set_covariance`**`(`*other*, *cov=None*`)`

Sets the covariance between this value and another value

The covariance between two variables is by default 0. Users can set the covariance between two measurements to any value, and it will be taken into account during error propagation. When two measurements are recorded as arrays of repeated measurements of the same length, users can leave the covariance term empty, and let QExPy calculate the covariance between them. You should only do this when these two quantities are measured at the same time, and can be related physically.

Examples

```
>>> import qexpy as q
>>> a = q.Measurement(5, 0.5)
>>> b = q.Measurement(6, 0.3)
```

```
>>> # The user can manually set the covariance between two values
>>> a.set_covariance(b, 0.135)
>>> a.get_covariance(b)
0.135
```

```
>>> # The correlation factor is calculated behind the scene as well
>>> a.get_correlation(b)
0.9
```

```
>>> # The user can ask QExPy to calculate the covariance if applicable
>>> a = q.Measurement([1, 1.2, 1.3, 1.4])
>>> b = q.Measurement([2, 2.1, 3, 2.3])
>>> a.set_covariance(b)  # this will declare that a and b are indeed correlated
>>> a.get_covariance(b)
0.0416667
```

`ExperimentalValue.`**`get_correlation`**`(`*other*`)`

Gets the correlation between this value and another value

Return type
`float`

`ExperimentalValue.`**`set_correlation`**`(`*other*, *corr=None*`)`

Sets the correlation between this value and another value

The correlation factor is a value between -1 and 1. This method can be used the same way as set_covariance.

See also:

`ExperimentalValue.set_covariance()`

CHAPTER

FIVE

THE MEASUREMENT OBJECT

To record values with an uncertainty, we use the `MeasuredValue` object. It is a child class of `ExperimentalValue`, so it inherits all attributes and methods from the `ExperimentalValue` class.

`class qexpy.data.data.`**`MeasuredValue`**(*data*, *error=None*, ***kwargs*)

Container for user-recorded values with uncertainties

The MeasuredValue represents a single measurement recorded in an experiment. This class is given an alias "Measurement" for backward compatibility and for a more intuitive user interface. On the top level of this package, this class is imported as "Measurement".

Parameters

- **`data`** (*`Real|List`*) – The center value of the measurement
- **`error`** (*`Real|List`*) – The uncertainty on the value

Keyword Arguments

- **`unit`** (*`str`*) – The unit of this value
- **`name`** (*`str`*) – The name of this value

5.1 Repeated Measurements

To record a value as the mean of a series of repeated measurements, use `RepeatedlyMeasuredValue`

`class qexpy.data.data.`**`RepeatedlyMeasuredValue`**(*data*, *error=None*, ***kwargs*)

Container for a MeasuredValue recorded as an array of repeated measurements

This class is instantiated if an array of values is used to record a Measurement of a single quantity with repeated takes. By default, the mean of the array is used as the value of this quantity, and the standard error (error on the mean) is the uncertainty. The reason for this choice is because the reason for taking multiple measurements is usually to minimize the uncertainty on the quantity, not to find out the uncertainty on a single measurement (which is what standard deviation is).

Examples

```
>>> import qexpy as q
```

```
>>> # The most common way of recording a value with repeated measurements is to only
>>> # give the center values for the measurements
>>> a = q.Measurement([9, 10, 11])
>>> print(a)
10.0 +/- 0.6
```

```
>>> # There are other statistical properties of the array of measurements
>>> a.std
1
>>> a.error_on_mean
0.5773502691896258
```

```
>>> # You can choose to use the standard deviation as the uncertainty
>>> a.use_std_for_uncertainty()
>>> a.error
1
```

```
>>> # You can also specify individual uncertainties for the measurements
>>> a = q.Measurement([10, 11], [0.1, 1])
>>> print(a)
10.5 +/- 0.5
>>> a.error_weighted_mean
10.00990099009901
>>> a.propagated_error
0.09950371902099892
```

```
>>> # You can choose which statistical properties to be used as the value/error
>>> a.use_error_weighted_mean_as_value()
>>> a.use_propagated_error_for_uncertainty()
>>> q.set_sig_figs_for_error(4)
>>> print(a)
10.00990 +/- 0.09950
```

5.1.1 Properties

`RepeatedlyMeasuredValue.`**`raw_data`**

The raw data that was used to generate this measurement

Type
np.ndarray

`RepeatedlyMeasuredValue.`**`mean`**

The mean of raw measurements

Type
float

`RepeatedlyMeasuredValue.`**`error_weighted_mean`**

Error weighted mean if individual errors are specified

Type
float

`RepeatedlyMeasuredValue.`**`std`**

The standard deviation of the raw data

Type
float

`RepeatedlyMeasuredValue.`**`error_on_mean`**

The error on the mean or the standard error

Type
float

`RepeatedlyMeasuredValue.`**`propagated_error`**

Error propagated with errors passed in if present

Type
float

5.1.2 Methods

`RepeatedlyMeasuredValue.`**`use_std_for_uncertainty`**`()`

Sets the uncertainty of this value to the standard deviation

`RepeatedlyMeasuredValue.`**`use_error_on_mean_for_uncertainty`**`()`

Sets the uncertainty of this value to the error on the mean

`RepeatedlyMeasuredValue.`**`use_error_weighted_mean_as_value`**`()`

Sets the value of this object to the error weighted mean

`RepeatedlyMeasuredValue.`**`use_propagated_error_for_uncertainty`**`()`

Sets the uncertainty of this object to the weight propagated error

`RepeatedlyMeasuredValue.`**`show_histogram`**`(`*`**kwargs`*`)`

Plots the raw measurement data in a histogram :rtype: `tuple`

See also:

This works the same as the `hist()` function in the plotting module of QExPy

5.2 Correlated Measurements

Sometimes in experiments, two measured quantities can be correlated, and this correlation needs to be accounted for during error propagation. QExPy provides methods that allows users to specify the correlation between two measurements, and it will be taken into account automatically during computations.

`qexpy.data.data.`**`set_correlation`**`(`*`var1`*`, `*`var2`*`, `*`corr=None`*`)`

Sets the correlation factor between two MeasuredValue objects

Parameters

- **`var1`** (`ExperimentalValue`) – the two values to set correlation between

- **var2** (ExperimentalValue) – the two values to set correlation between

See also:

ExperimentalValue.set_correlation()

`qexpy.data.data.`**`get_correlation`**`(`*var1*, *var2*`)`

Finds the correlation between two ExperimentalValue instances

Parameters

- **var1** (ExperimentalValue) – the two values to find correlation between
- **var2** (ExperimentalValue) – the two values to find correlation between

Return type

`float`

Returns

The correlation factor between var1 and var2

See also:

ExperimentalValue.get_correlation()

`qexpy.data.data.`**`set_covariance`**`(`*var1*, *var2*, *cov=None*`)`

Sets the covariance between two measurements

Parameters

- **var1** (ExperimentalValue) – the two values to set covariance between
- **var2** (ExperimentalValue) – the two values to set covariance between

See also:

ExperimentalValue.set_covariance()

Examples

```
>>> import qexpy as q
>>> a = q.Measurement(5, 0.5)
>>> b = q.Measurement(6, 0.3)
```

```
>>> # The user can manually set the covariance between two values
>>> q.set_covariance(a, b, 0.135)
>>> q.get_covariance(a, b)
0.135
```

`qexpy.data.data.`**`get_covariance`**`(`*var1*, *var2*`)`

Finds the covariances between two ExperimentalValue instances

Parameters

- **var1** (ExperimentalValue) – the two values to find covariance between
- **var2** (ExperimentalValue) – the two values to find covariance between

Return type

`float`

Returns

The covariance between var1 and var2

See also:

`ExperimentalValue.get_covariance()`

There are also shortcuts to the above methods implemented in *`ExperimentalValue`*.

`MeasuredValue.`**`set_correlation`**(*other*, *corr=None*)

Sets the correlation factor of this value with another value

`MeasuredValue.`**`get_correlation`**(*other*)

Gets the correlation factor of this value with another value

Return type
`float`

`MeasuredValue.`**`set_covariance`**(*other*, *cov=None*)

Sets the covariance of this value with another value

`MeasuredValue.`**`get_covariance`**(*other*)

Gets the covariance of this value with another value

Return type
`float`

CHAPTER

SIX

THE MEASUREMENTARRAY OBJECT

Using QExPy, the user is able to record a series of measurements, and store them in an array. This feature is implemented in QExPy as a wrapper around `numpy.ndarray`. The *ExperimentalValueArray* class, also given the alias `MeasurementArray` stores an array of values with uncertainties, and it also comes with methods for some basic data processing.

class qexpy.data.datasets.**ExperimentalValueArray**(**args*, ***kwargs*)

An array of experimental values, alias: MeasurementArray

An ExperimentalValueArray (MeasurementArray) represents a series of ExperimentalValue objects. It is implemented as a sub-class of numpy.ndarray. This class is given an alias "MeasurementArray" for more intuitive user interface.

Parameters

***args** – The first argument is an array of real numbers representing the center values of the measurements. The second argument (if present) is either a positive real number or an array of positive real numbers of the same length as the data array, representing the uncertainties on the measurements.

Keyword Arguments

- **data** (*List*) – an array of real numbers representing the center values
- **error** (*Real | List*) – the uncertainties on the measurements
- **relative_error** (*Real | List*) – the relative uncertainties on the measurements
- **unit** (*str*) – the unit of the measurement
- **name** (*str*) – the name of the measurement

Examples

```
>>> import qexpy as q
```

```
>>> # We can instantiate an array of measurements with two lists
>>> a = q.MeasurementArray([1, 2, 3, 4, 5], [0.1, 0.2, 0.3, 0.4, 0.5])
>>> a
ExperimentalValueArray([MeasuredValue(1.0 +/- 0.1),
            MeasuredValue(2.0 +/- 0.2),
            MeasuredValue(3.0 +/- 0.3),
            MeasuredValue(4.0 +/- 0.4),
            MeasuredValue(5.0 +/- 0.5)], dtype=object)
```

```
>>> # We can also create an array of measurements with a single uncertainty.
>>> # As usual, if the error is not specified, they will be set to 0 by default
>>> a = q.MeasurementArray([1, 2, 3, 4, 5], 0.5, unit="m", name="length")
>>> a
ExperimentalValueArray([MeasuredValue(1.0 +/- 0.5),
            MeasuredValue(2.0 +/- 0.5),
            MeasuredValue(3.0 +/- 0.5),
            MeasuredValue(4.0 +/- 0.5),
            MeasuredValue(5.0 +/- 0.5)], dtype=object)
```

```
>>> # We can access the different statistical properties of this array
>>> print(np.sum(a))
15 +/- 1 [m]
>>> print(a.mean())
3.0 +/- 0.7 [m]
>>> a.std()
1.5811388300841898
```

```
>>> # Manipulation of a MeasurementArray is also very easy. We can append or insert
>>> # into the array values in multiple formats
>>> a = a.append((7, 0.2))  # a measurement can be inserted as a tuple
>>> print(a[5])
length = 7.0 +/- 0.2 [m]
>>> a = a.insert(0, 8)  # if error is not specified, it is set to 0 by default
>>> print(a[0])
length = 8 +/- 0 [m]
```

```
>>> # The same operations also works with array-like objects, in which case they are
>>> # concatenated into a single array
>>> a = a.append([(10, 0.1), (11, 0.3)])
>>> a
ExperimentalValueArray([MeasuredValue(8.0 +/- 0),
            MeasuredValue(1.0 +/- 0.5),
            MeasuredValue(2.0 +/- 0.5),
            MeasuredValue(3.0 +/- 0.5),
            MeasuredValue(4.0 +/- 0.5),
            MeasuredValue(5.0 +/- 0.5),
            MeasuredValue(7.0 +/- 0.2),
            MeasuredValue(10.0 +/- 0.1),
            MeasuredValue(11.0 +/- 0.3)], dtype=object)
```

```
>>> # The ExperimentalValueArray object is vectorized just like numpy.ndarray. You
>>> # can perform basic arithmetic operations as well as functions with them and get
>>> # back ExperimentalValueArray objects
>>> a = q.MeasurementArray([0, 1, 2], 0.5)
>>> a + 2
ExperimentalValueArray([DerivedValue(2.0 +/- 0.5),
                DerivedValue(3.0 +/- 0.5),
                DerivedValue(4.0 +/- 0.5)], dtype=object)
>>> q.sin(a)
ExperimentalValueArray([DerivedValue(0.0 +/- 0.5),
                DerivedValue(0.8 +/- 0.3),
```

(continues on next page)

(continued from previous page)

```
                 DerivedValue(0.9 +/- 0.2)], dtype=object)
```

See also:

numpy.ndarray

6.1 Properties

`ExperimentalValueArray.`**`values`**

An array consisting of the center values of each item

Type
np.ndarray

`ExperimentalValueArray.`**`errors`**

An array consisting of the uncertainties of each item

Type
np.ndarray

`ExperimentalValueArray.`**`name`**

Name of this array of values

A name can be given to this data set, and each measurement within this list will be named in the form of "name_index". For example, if the name is specified as "length", the items in this array will be named "length_0", "length_1", "length_2", . . .

Type
str

`ExperimentalValueArray.`**`unit`**

The unit of this array of values

It is assumed that the set of data that constitutes one ExperimentalValueArray have the same unit, which, when assigned, is given too all the items of the array.

Type
str

6.2 Methods

`ExperimentalValueArray.`**`mean`**`(**_)`

The mean of the array

Return type
`ExperimentalValue`

`ExperimentalValueArray.`**`std`**`(`*ddof=1*`, **_)`

The standard deviation of this array

Return type
`float`

`ExperimentalValueArray.`**`sum`**`(**_)`

The sum of the array

Return type
ExperimentalValue

`ExperimentalValueArray.`**`error_on_mean`**`()`

The error on the mean of this array

Return type
`float`

`ExperimentalValueArray.`**`error_weighted_mean`**`()`

The error weighted mean of this array

Return type
`float`

`ExperimentalValueArray.`**`propagated_error`**`()`

The propagated error from the error weighted mean calculation

Return type
`float`

`ExperimentalValueArray.`**`append`**`(`*value*`)`

Adds a value to the end of this array and returns the new array

Parameters
value – The value to be appended to this array. This can be a real number, a pair of value and error in a tuple, an ExperimentalValue instance, or an array consisting of any of the above.

Return type
ExperimentalValueArray

Returns
The new ExperimentalValueArray instance

`ExperimentalValueArray.`**`delete`**`(`*index*`)`

deletes the value on the requested position and returns the new array

Parameters
index (*int*) – the index of the value to be deleted

Return type
ExperimentalValueArray

Returns
The new ExperimentalValueArray instance

`ExperimentalValueArray.`**`insert`**`(`*index*, *value*`)`

adds a value to a position in this array and returns the new array

Parameters

- **index** (*int*) – the position to insert the value
- **value** – The value to be inserted into this array. This can be a real number, a pair of value and error in a tuple, an ExperimentalValue instance, or an array consisting of any of the above.

Return type
ExperimentalValueArray

Returns

The new ExperimentalValueArray instance

CHAPTER

SEVEN

ERROR PROPAGATION

Error propagation is implemented as a child class of `ExperimentalValue` called `DerivedValue`. When working with QExPy, the result of all computations are stored as instances of this class.

7.1 The DerivedValue Object

class qexpy.data.data.**DerivedValue**(*formula*)

Result of calculations performed with ExperimentalValue instances

This class is automatically instantiated when the user performs calculations with other ExperimentalValue instances. It is created with the properly propagated uncertainties and units. The two available methods for error propagation are the derivative method, and the Monte Carlo method.

Internally, a DerivedValue preserves information on how it is calculated, so the user is able to make use of that information. For example, the user can find the derivative of a DerivedValue with respect to another ExperimentalValue that this value is derived from.

Examples

```
>>> import qexpy as q
```

```
>>> # First let's create some standard measurements
>>> a = q.Measurement(5, 0.2)
>>> b = q.Measurement(4, 0.1)
>>> c = q.Measurement(6.3, 0.5)
>>> d = q.Measurement(7.2, 0.5)
```

```
>>> # Now we can perform operations on them
>>> result = q.sqrt(c) * d - b / q.exp(a)
>>> result
DerivedValue(18 +/- 1)
>>> result.value
18.04490478513969
>>> result.error
1.4454463754287323
```

```
>>> # By default, the standard derivative method is used, but it can be changed
>>> q.set_error_method(q.ErrorMethod.MONTE_CARLO)
```

(continues on next page)

(continued from previous page)

```
>>> result.value
18.03203135268583
>>> result.error
1.4116412532654283
>>> # If we want this value to use a different error method from the global default
>>> result.error_method = "derivative" # this only affects this value alone
>>> result.error
1.4454463754287323
>>> # If we want to reset the error method for this value and use the global default
>>> result.reset_error_method()
>>> result.error
1.4116412532654283
```

7.1.1 Properties

`DerivedValue.`**`value`**

`DerivedValue.`**`error`**

`DerivedValue.`**`relative_error`**

`DerivedValue.`**`error_method`**

The default error method used for this value

QExPy currently supports two different methods of error propagation, the derivative method, and the Monte-Carlo method. The user can change the global default which applies to all values, or set the error method of this single quantity if it is to be different from the global settings.

Type
ErrorMethod

`DerivedValue.`**`mc`**

The settings object for customizing Monte Carlo

Type
dut.MonteCarloSettings

7.1.2 Methods

`DerivedValue.`**`reset_error_method`**`()`

Resets the default error method for this value to follow the global settings

`DerivedValue.`**`recalculate`**`()`

Recalculates the value

A DerivedValue instance preserves information on how the value was derived. If values of the original measurements are changed, and you wish to update the derived value using the exact same formula, this method can be used.

Examples

```
>>> import qexpy as q
```

```
>>> a = q.Measurement(5, 0.2)
>>> b = q.Measurement(4, 0.1)
```

```
>>> c = a + b
>>> c
DerivedValue(9.0 +/- 0.2)
```

```
>>> # Now we change the value of a
>>> a.value = 8
>>> c.recalculate()
>>> c
DerivedValue(12.0 +/- 0.2)
```

`DerivedValue.`**`show_error_contributions`**`()`

Displays measurements' contribution to the final uncertainty

7.2 The MonteCarloSettings Object

QExPy provides users with many options to customize Monte Carlo error propagation. Each `DerivedValue` object stores a `MonteCarloSettings` object that contains some settings for the Monte Carlo error propagation of this value.

`class` `qexpy.data.utils.`**`MonteCarloSettings`**`(`*evaluator*`)`

The object for customizing the Monte Carlo error propagation process

7.2.1 Properties

`MonteCarloSettings.`**`sample_size`**

The Monte Carlo sample size

Type
int

`MonteCarloSettings.`**`confidence`**

The confidence level for choosing the mode of a Monte Carlo distribution

Type
float

`MonteCarloSettings.`**`xrange`**

The x-range of the simulation

This is really the y-range, which means it's the range of the y-values to show, but also this is the x-range of the histogram.

Type
tuple

7.2.2 Methods

`MonteCarloSettings.`**`set_xrange`**`(`*`*args`*`)`

set the range for the monte carlo simulation

`MonteCarloSettings.`**`use_mode_with_confidence`**`(`*`confidence=None`*`)`

Use the mode of the distribution with a confidence coverage for this value

`MonteCarloSettings.`**`use_mean_and_std`**`()`

Use the mean and std of the distribution for this value

`MonteCarloSettings.`**`show_histogram`**`(`*`bins=100`*`, `*`**kwargs`*`)`

Shows the distribution of the Monte Carlo simulated samples

`MonteCarloSettings.`**`samples`**`()`

The raw samples generated in the Monte Carlo simulation

Sometimes when the distribution is not typical, you might wish to do your own analysis with the raw samples generated in the Monte Carlo simulation. This method allows you to access a copy of the raw data.

`MonteCarloSettings.`**`use_custom_value_and_error`**`(`*`value`*`, `*`error`*`)`

Manually set the value and uncertainty for this quantity

Sometimes when the distribution is not typical, and you wish to see for yourself what the best approach is to choose the center value and uncertainty for this quantity, use this method to manually set these values.

CHAPTER

EIGHT

THE XYDATASET OBJECT

class qexpy.data.**XYDataSet**(**args*, ***kwargs*)

A pair of ExperimentalValueArray objects

QExPy is capable of multiple ways of data handling. One typical case in experimental data analysis is for a pair of data sets, which is usually plotted or fitted with a curve.

Parameters

- **xdata** (*List|np.ndarray*) – an array of values for x-data
- **ydata** (*List|np.ndarray*) – an array of values for y-data

Keyword Arguments

- **xerr** (*Real|List*) – the uncertainty on x data
- **yerr** (*Real|List*) – the uncertainty on y data
- **xunit** (*str*) – the unit of the x data set
- **yunit** (*str*) – the unit of the y data set
- **xname** (*str*) – the name of the x data set
- **yname** (*str*) – the name of the y data set

Examples

```
>>> import qexpy as q
```

```
>>> a = q.XYDataSet(xdata=[0, 1, 2, 3, 4], xerr=0.5, xunit="m", xname="length",
>>>                 ydata=[3, 4, 5, 6, 7], yerr=[0.1,0.2,0.3,0.4,0.5],
>>>                 yunit="kg", yname="weight")
>>> a.xvalues
array([0, 1, 2, 3, 4])
>>> a.xerr
array([0.5, 0.5, 0.5, 0.5, 0.5])
>>> a.yerr
array([0.1, 0.2, 0.3, 0.4, 0.5])
>>> a.xdata
ExperimentalValueArray([MeasuredValue(0.0 +/- 0.5),
                MeasuredValue(1.0 +/- 0.5),
                MeasuredValue(2.0 +/- 0.5),
                MeasuredValue(3.0 +/- 0.5),
                MeasuredValue(4.0 +/- 0.5)], dtype=object)
```

8.1 Properties

`XYDataSet.`**`xvalues`**

The values of the x data set

Type
np.ndarray

`XYDataSet.`**`xerr`**

The errors of the x data set

Type
np.ndarray

`XYDataSet.`**`yvalues`**

The values of the y data set

Type
np.ndarray

`XYDataSet.`**`yerr`**

The errors of the x data set

Type
np.ndarray

`XYDataSet.`**`xname`**

Name of the xdata set

Type
str

`XYDataSet.`**`yname`**

Name of the ydata set

Type
str

`XYDataSet.`**`xunit`**

Unit of the xdata set

Type
str

`XYDataSet.`**`yunit`**

Unit of the ydata set

Type
str

8.2 Methods

`XYDataSet.`**`fit`**`(`*model*, ***kwargs*`)`

Fits the current dataset to a model

See also:

The fit function in the fitting module of QExPy

CHAPTER

NINE

THE FITTING MODULE

`qexpy.fitting.`**`fit`**(**args*, ***kwargs*)

Perform a fit to a data set

The fit function can be called on an XYDataSet object, or two arrays or MeasurementArray objects. QExPy provides 5 builtin fit models, which includes linear fit, quadratic fit, general polynomial fit, gaussian fit, and exponential fit. The user can also pass in a custom function they wish to fit their dataset on. For non-polynomial fit functions, the user would usually need to pass in an array of guesses for the parameters.

Parameters

`*args` – An XYDataSet object or two arrays to be fitted.

Keyword Arguments

- **`model`** – the fit model given as the string or enum representation of a pre-set model or a custom callable function with parameters. Available pre-set models include: "linear", "quadratic", "polynomial", "exponential", "gaussian"
- **`xrange`** (*`tuple | list`*) – a pair of numbers indicating the domain of the function
- **`degrees`** (*`int`*) – the degree of the polynomial if polynomial fit were chosen
- **`parguess`** (*`list`*) – initial guess for the parameters
- **`parnames`** (*`list`*) – the names of each parameter
- **`parunits`** (*`list`*) – the units for each parameter
- **`dataset`** – the XYDataSet instance to fit on
- **`xdata`** – the x-data of the fit
- **`ydata`** – the y-data of the fit
- **`xerr`** – the uncertainty on the xdata
- **`yerr`** – the uncertainty on the ydata

Returns

the result of the fit

Return type

XYFitResult

See also:

`XYDataSet`

9.1 The XYFitResult Class

`class qexpy.fitting.fitting.`**`XYFitResult`**`(**kwargs)`

Stores the results of a curve fit

`XYFitResult.`**`dataset`**

The dataset used for this fit

Type
dts.XYDataSet

`XYFitResult.`**`fit_function`**

The function that fits to this data set

Type
Callable

`XYFitResult.`**`params`**

The fit parameters of the fit function

Type
List[dt.ExperimentalValue]

`XYFitResult.`**`residuals`**

The residuals of the fit

Type
dts.ExperimentalValueArray

`XYFitResult.`**`chi_squared`**

The goodness of fit represented as chi^2

Type
dt.ExperimentalValue

`XYFitResult.`**`ndof`**

The degree of freedom of this fit function

Type
int

`XYFitResult.`**`xrange`**

The xrange of the fit

Type
tuple

CHAPTER TEN

THE PLOTTING MODULE

`qexpy.plotting.plotting.plot(*args, **kwargs)`

Plots a dataset or a function

Adds a dataset or a function to a Plot, and returns the Plot object. This is a wrapper around the matplotlib.pyplot.plot function, so it takes all the keyword arguments that is accepted by the pyplot.plot function, as well as the pyplot.errorbar function.

By default, error bars are not displayed. If you want error bars, it can be turned on in the Plot object.

Parameters

`*args` – The first arguments can be an XYDataSet object, two separate arrays for xdata and ydata, a callable function, or an XYFitResult object. The function also takes a string at the end of the list of arguments as the format string.

Keyword Arguments

- **`xdata`** – a list of data for x-values
- **`xerr`** – the uncertainties for the x-values
- **`ydata`** – a list of data for y-values
- **`yerr`** – the uncertainties for the y-values
- **`xrange`** (*`tuple`*) – a tuple of two values specifying the x-range for the data to plot
- **`xname`** (*`str`*) – the name of the x-values
- **`yname`** (*`str`*) – the name of the y-values
- **`xunit`** (*`str`*) – the unit of the x-values
- **`yunit`** (*`str`*) – the unit of the y-values
- **`fmt`** (*`str`*) – the format string for the object to be plotted (matplotlib style)
- **`color`** (*`str`*) – the color for the object to be plotted
- **`label`** (*`str`*) – the label for the object to be displayed in the legend
- **`**kwargs`** – additional keyword arguments that matplotlib.pyplot.plot supports

Return type

`Plot`

See also:

`XYDataSet`, pyplot.plot, pyplot.errorbar

`qexpy.plotting.plotting.`**`hist`**`(`**args*, ***kwargs*`)`

Plots a histogram with a data set

Parameters
`*args` – the ExperimentalValueArray or arguments that creates an ExperimentalValueArray

Return type
`tuple`

See also:

hist()

`qexpy.plotting.plotting.`**`show`**`(`*plot_obj=None*`)`

Draws the plot to output

The QExPy plotting module keeps a buffer on the last plot being operated on. If no Plot instance is supplied to this function, the buffered plot will be shown.

Parameters
`plot_obj` (`Plot`) – the Plot instance to be shown.

`qexpy.plotting.plotting.`**`savefig`**`(`*filename*, *plot_obj=None*, ***kwargs*`)`

Save the plot into a file

The QExPy plotting module keeps a buffer on the last plot being operated on. If no Plot instance is supplied to this function, the buffered plot will be shown.

Parameters

- **`filename`** (*`string`*) – name and format of the file (ex: myplot.pdf),
- **`plot_obj`** (`Plot`) – the Plot instance to be shown.

`qexpy.plotting.plotting.`**`get_plot`**`()`

Gets the current plot buffer

`qexpy.plotting.plotting.`**`new_plot`**`()`

Clears the current plot buffer and start a new one

10.1 The Plot Object

`class` `qexpy.plotting.plotting.`**`Plot`**

The data structure used for a plot

10.1.1 Properties

`Plot.`**`title`**

The title of this plot, which will appear on top of the figure

Type
str

`Plot.`**`xname`**

The name of the x data, which will appear as x label

Type
str

`Plot.`**`yname`**

The name of the y data, which will appear as y label

Type
str

`Plot.`**`xunit`**

The unit of the x data, which will appear on the x label

Type
str

`Plot.`**`yunit`**

The unit of the y data, which will appear on the y label

Type
str

`Plot.`**`xlabel`**

The xlabel of the plot

Type
str

`Plot.`**`ylabel`**

the ylabel of the plot

Type
str

`Plot.`**`xrange`**

The x-value domain of this plot

Type
tuple

10.1.2 Methods

`Plot.`**`plot`**`(`**args*, ***kwargs*`)`

Adds a data set or function to the plot

See also:

`plot()`

`Plot.`**`hist`**`(`**args*, ***kwargs*`)`

Adds a histogram to the plot

See also:

`hist()`

`Plot.`**`fit`**`(`**args*, ***kwargs*`)`

Plots a curve fit to the last data set added to the figure

The fit function finds the last data set or histogram added to the Plot and apply a fit to it. This function takes the same arguments as QExPy fit function, and the same keyword arguments as in the QExPy plot function in configuring how the line of best fit shows up on the plot.

See also:

fit() *plot()*

Plot.**show**()

Draws the plot to output

Plot.**legend**(*new_setting=True*)

Add or remove legend to plot

Plot.**error_bars**(*new_setting=True*)

Add or remove error bars from plot

Plot.**residuals**(*new_setting=True*)

Add or remove subplot to show residuals

Plot.**savefig**(*filename*, ***kwargs*)

Save figure using matplotlib

CHAPTER ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX